



GPGPU for Difficult Black-box Problems

Marcin Pietron, Aleksander Byrski, Marek Kisiel-Dorohinicki

AGH University of Science and Technology
Faculty of Computer Science, Electronics and Telecommunications
Al. Mickiewicza 30, 30-962, Krakow, Poland
{pietron,olekb,doroh}@agh.edu.pl

Abstract

Difficult black-box problems arise in many scientific and industrial areas. In this paper, efficient use of a hardware accelerator to implement dedicated solvers for such problems is discussed and studied based on an example of Golomb Ruler problem. The actual solution of the problem is shown based on evolutionary and memetic algorithms accelerated on GPGPU. The presented results prove that GPGPU outperforms CPU in some memetic algorithms which can be used as a part of hybrid algorithm of finding near optimal solutions of Golomb Ruler problem. The presented research is a part of building heterogenous parallel algorithm for difficult black-box Golomb Ruler problem.

Keywords: Evolutionary Computing, GPGPU computing, memetic computing

1 Introduction

Low Autocorrelation Binary Sequences, Job Shop, Golomb Ruler are well known difficult combinatorial problems [4]. They are good examples of so-called black-box scenarios [10] posing serious problem for the solving software, demanding application of metaheuristic approach in order to locate even sub-optimal solutions.

The most popular metaheuristics used for such problems are dynamic programming, constraint programming, simulated annealing, tabu search, evolutionary and memetic algorithms (see, e.g. [13], [22], [5], [8], [9]).

The hybrids of evolutionary and local search algorithms are very often used in solving NP-hard problems [6]. The parallel hardware platforms (e.g. graphic cards, multicore processors) enable to make use of data flow structure (with contradiction to other heuristics like constraint or dynamic programming) of evolutionary and memetic algorithms and can help to improve efficiency of solving difficult problems [11] making possible development of dedicated software environments, taking advantage of their specific features. Interesting possibilities arise when using dedicated hardware is considered, such as graphical processing units (GPGPUs), which enable to run thousands of threads in parallel. The top peak performance of the most efficient high performance graphic card is over 1 TB/s. However, the programmer or designer must be aware

of memory hierarchy which has significant influence on algorithm (see Section 2). A significant number of numerical and data mining algorithms were efficiently implemented using GPGPUs [16], [15], [20], [19], [17].

In this work we study ability of parallelizing of metaheuristics in GPGPU platform choosing a very hard combinatorial problem, namely Optimal Golomb Ruler, as a case study. The examples of using Golomb Rulers can be found in Information Theory related to error correcting codes, selection of radio frequencies to reduce the effects of intermodulation interference with both terrestrial and extraterrestrial applications or design of phased arrays of radio antennas or in astronomy one-dimensional synthesis arrays.

In this paper we focus firstly on GPGPU architecture and its parallel characteristics. Later we deal with details of GPGPU implementation for evolutionary solving for Golomb ruler. In the end the experimental results are shown and the paper is concluded.

2 GPGPU and multiprocessor computing

The architecture of a GPGPU card is described in Fig. 1. GPGPU is constructed as N multiprocessor structure with M cores each. The cores share an Instruction Unit with other cores in a multiprocessor. Multiprocessors have dedicated memory chips which are much faster than global memory, shared for all multiprocessors. These memories are: read-only constant/texture memory and shared memory. The GPGPU cards are constructed as massive parallel devices, enabling thousands of parallel threads to run which are grouped in blocks with shared memory. A dedicated software architecture—CUDA—makes possible programming GPGPU using high-level languages such as C and C++ [1]. CUDA requires an NVIDIA GPGPU like Fermi, GeForce 8XXX/Tesla/Quadro etc. This technology provides three key mechanisms to parallelize programs: thread group hierarchy, shared memories, and barrier synchronization. These mechanisms provide fine-grained parallelism nested within coarse-grained task parallelism. Creating the optimized code is not trivial and thorough knowledge of GPGPUs architecture is necessary to do it effectively. The main aspects to consider are the usage of the memories, efficient division of code into parallel threads and thread communications. As it was mentioned earlier, constant/texture, shared memories and local memories are specially optimized regarding the access time, therefore programmers should optimally use them to speedup access to data on which an algorithm operates. Another important thing is to optimize synchronization and the communication of the threads. The synchronization of the threads between blocks is much slower than in a block. If it is not necessary it should be avoided, if necessary, it should be solved by the sequential running of multiple kernels. Another important aspect is the fact that recursive function calls are not allowed in CUDA kernels. Providing stack space for all the active threads requires substantial amounts of memory.

Modern processors consist of two or more independent central processing units. This architecture enables multiple CPU instructions (add, move data, branch etc.) to run at the same time. The cores are integrated into a single integrated circuit. The manufacturers AMD and Intel have developed several multi-core processors (dual-core, quad-core, hexa-core, octa-core etc.). The cores may or may not share caches, and they may implement message passing or shared memory inter-core communication. The single cores in multi-core systems may implement architectures such as vector processing, SIMD, or multi-threading. These techniques offer another aspect of parallelization (implicit to high level languages, used by compilers). The performance gained by the use of a multi-core processor depends on the algorithms used and their implementation. The multicore-processors are used for comparison of the developed GPGPU computing solution discussed in this paper, so experiments run on single-core and multi-core

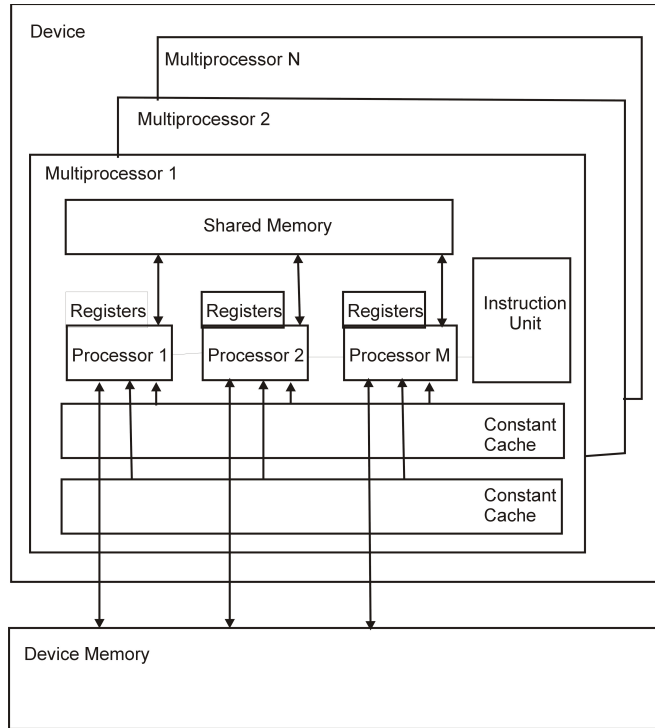


Figure 1: GPGPU architecture

processors will be used to show the scalability of evolutionary and memetic algorithms in CPU. It is to note, that in both cases (GPGPU and CPU computing) the code is optimized and appropriately modified in order to exploit all possible features of both solutions.

There are lot of programming models and libraries of multi-core programming. The most popular are pthreads, OpenMP, Cilk++, TDD etc. In our work OpenMP was used [3], being a software platform supporting multi-threaded, shared-memory parallel processing multi-core architectures for C, C++ and Fortran languages. By using OpenMP, the programmer does not need to create the threads nor assign tasks to each thread. The programmer inserts directives to assist the compiler into generating threads for the parallel processor platform.

3 Golomb Ruler black-box problem

The Golomb Ruler is a set of marks at integer positions along an imaginary ruler such that no two pairs of marks are the same distance apart. The number of marks on the ruler is its order, and the largest distance between two of its marks is its length (distance between last and the first element). Golomb Ruler is optimal if no shorter GR of the same order exists. Fig. 2 shows optimal golomb ruler with 5 marks. The process of creating GR is easy, but finding the optimal OGR (or rulers) for a specified order is computationally very challenging. For instance, the search for 19 marks of GR took approximately 36200 CPU hours on a Sun Sparc workstation using brute force parallel search implementation. The best known maximal length GR found up-to-now is 27.

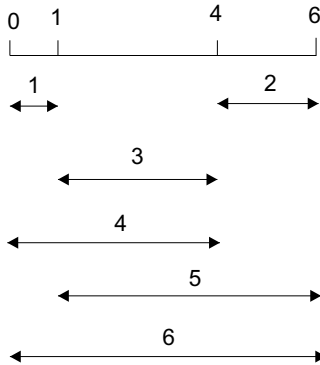


Figure 2: Structure of a Golomb Ruler

A well-known sequential solution of OGR problem was proposed by Shearer [21]. It is based on branch and bound algorithm with backtracking. It generates optimal GRs up to 16. Soliday [22] proposed an algorithm in which chromosomes are represented by integers having length of $n - 1$ segments, mutation operator is composed of two types: a change in the segment length or a permutation in the segment order. They use two evaluation criteria such as the overall length of the ruler and the number of repeated measurements.

First hybrid approaches of evolutionary algorithms were introduced by Feeney [12]. It combines genetic algorithms with local search technique and Baldwinian or Lamarckian learning [7]. The proposed representation consists of an array of integers corresponding to the marks. The crossover operator is a random swap between two positions and a sort procedure was added at the end. The distance achieved from optimal rulers is between 6.8 and 20.3. Van Henteryck and Dotu [9] created evolutionary algorithm with Tabu Search in mutation operator and single-point crossover. The algorithm uses a random strategy in selection process to choose parents for breeding (distances from optimal for 12 to 16 marks rulers are between 7.1 and 10.2). Cotta, Dota and Van Henteryck used GRASP (greedy randomized adaptive search) method, scatter search and tabu search, clustering techniques and constraint programming. Combining these techniques memetic algorithm was proposed. The distances to the optimum between 10 to 16 length of OGR were between 1.6 and 6.2 [9]. Parallel solutions [14] and [2] were able to find optimal rulers up to 26 marks in several months on thousands computers.

4 Implementation of GPGPU for evolutionary solving of Golomb Ruler

The general purpose graphic cards are commonly used as computing accelerators in many scientific problems. The image processing, data mining, numerical algorithms are most popular domains in which GPGPU were used with a success. Recently we can observe that also computational intelligence, multi-objective optimization make use of graphic cards architecture and computing power. The adaptation of each algorithm must be preceded by careful analysis, concerning data flow in the particular algorithms (data dependence), extracting hidden parallelism

and appropriate data mapping to device memory hierarchy. Such studies show that evolutionary algorithms are better to adapt to parallel hardware accelerators than other metaheuristics like dynamic programming, constraint programming etc. Almost each of the usually realized steps can be parallelized in evolutionary algorithm (crossover operator, mutation operator run on whole population etc.). The challenging problem in adaptation the evolutionary algorithms in GPGPU is data mapping. The each problem can have its own representation. Moreover, fitness function can be calculated in several ways. These artefacts have strong influence on data size and data usage of the algorithm.

The evolutionary algorithm can be realized on a host utilizing GPGPU in a hybrid way. At the first step, initial population is created on the host, then population is sent to GPGPU to realize crossover operator and memetic algorithm (see Fig. 3).

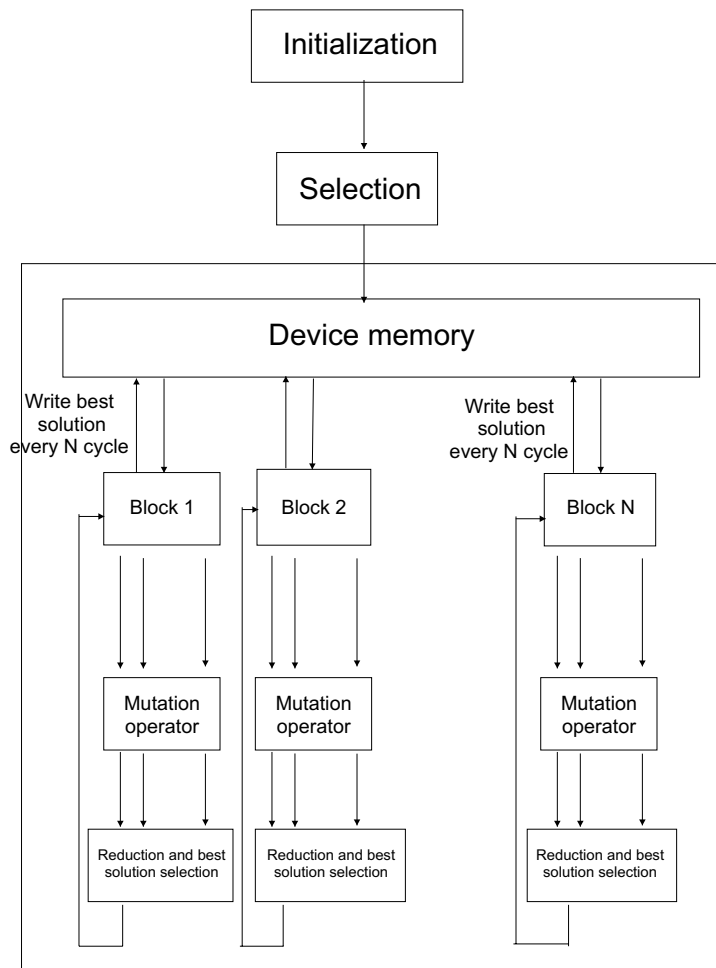


Figure 3: The hybrid algorithm of solving Golomb ruler.

Memetic algorithms can be also be designed to make possible parallelization of their parts. Our main goal was to find heuristics for solving Golomb Ruler problem that can be accelerated in GPGPUs and compare them with highly optimized counterparts implemented in CPU. The CPU implementations are C-based, fully vectorized and also multi-core adapted. The approaches similar to Random Hill Climbing (RMHC) and Simulated Annealing was proposed with some specific modifications. In our implementation memetic algorithm of solving Golomb Ruler Problem was constructed to exploit computational capabilities and memory bandwidth of GPGPU as much as possible. The shared and local memory data optimization was done by minimizing following parameters: data space for storing genotypes, number of shared memory bank conflicts (see Fig. 5 and Fig. 6).

Our implementation enables running mutation, crossover operators and memetic part of the evolutionary algorithm in GPGPU. The crossover operators are implemented in two ways as single and double point version. In both operators *curand* library is used for choosing random points in a single representation and then copy genomes between two solutions in case crossover operator and generate random changes in selected genome in case of mutation (see Fig. 4). The first kernel is responsible for generating random numbers, each thread generates one random number which belongs to interval $[1, size_of_representation]$. In the second kernel each thread mutates or copies genomes between crossed-over representations. The last step is responsible for fitness computation based on changes from genetic operators.

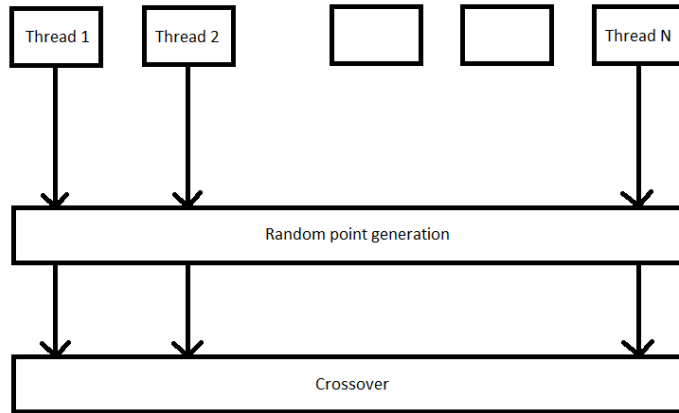


Figure 4: The architecture of crossover operator.

The whole algorithm implemented in GPGPU is described in Fig. 3. It exploits GPGPUs computational resources as much as possible. The kernel is run on number of blocks equals size of population. Each block copies separate solution from the global memory and its histogram of all possible ruler's differences. Next, all available threads in the block run mutation operator on the solution stored in their shared memory. Each thread generates two random numbers in mutation process. The first one is point in golomb ruler to be mutated and the second one is new value on that position.

The algorithm needs $number_of_threads_in_block \cdot 4 \cdot (size_of_ruler - 1) \cdot 4$ bytes of shared memory for data storing new and old differences between chosen position and other points. From this data fitness value is computed for each mutated solution (see Fig. 5), the elements with even indexes in table are values added or removed from histogram after mutation and

elements with odd indexes are number of value of modification of these values in histogram. The values needed for updating the histogram and computing the fitness are stored in a manner to minimize bank conflicts (see Fig. 5). The fitness values and index of the thread are written to the shared memory for a reduction process ($1024 \cdot 2 \cdot 4$ bytes) which will be run for choosing best mutated solution (see Fig. 6). Two parallel reduction are run. The first one for finding best fitness value and the second one for finding which solution is representing the best temporal ruler. Between iterations in blocks, the threads can exchange the information about optimal solution by temporary minimal ruler found in each block. It can be done by atomic operation on global memory variable (atomicMin CUDA function). This process can help to narrow domain space of possible optimal solutions (greedy algorithm) in next iterations. In case of parallel implementation in multi-core environment openmp directives were used. The outer loop of algorithm (responsible for iteration over solutions) was parallelized. The OpenMP lock mechanism was used for updating the global temporal minimal ruler found.

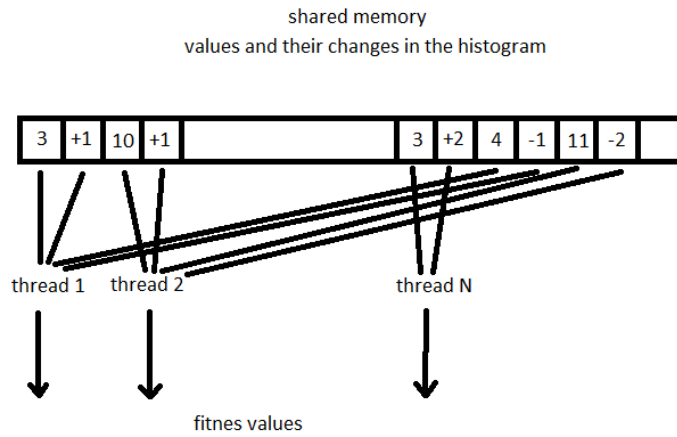


Figure 5: The histogram update computation.

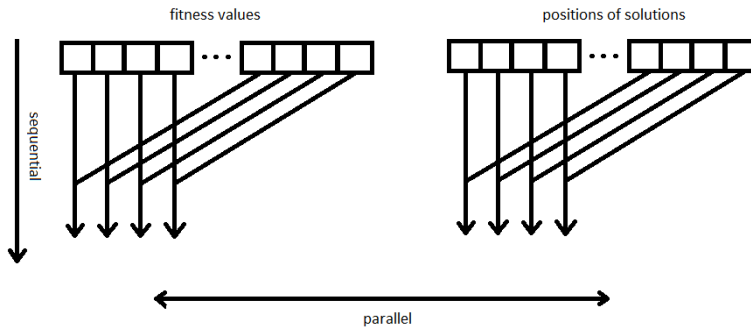


Figure 6: Reduction process of finding best solution and its position

Table 1: Execution times (in ms) of algorithm for Golomb ruler 6 (80 iterations of algorithm).

Population size (GR 6)	GPGPU	CPU (1 core)	CPU (-O3) (1 core)	4-cores (-O3)
64	18	677	260	83
128	38	1363	556	170
256	94	2823	1103	380
512	255	5685	2202	730

Table 2: Execution times of execution (in ms) of algorithm for Golomb ruler 7 (80 iterations of algorithm).

Population size (GR 7)	GPGPU	CPU (1 core)	CPU (-O3) (1 core)	4-cores (-O3)
64	30	987	380	140
128	51	1850	870	280

5 Experimental results

The results presented in Table 1, Table 2 and Table 3 show the execution times of the presented algorithm. It was run memetic algorithm consisting in hybridization of evolutionary algorithm with a single-point random mutation hill climbing [18] local search operator. Each iteration consisted of generating 512 new mutated solutions from single solution. Every each 20 cycles one point crossover were run in GPGPU. in GPGPU, normal single core CPU, vectorized and highly optimized single core and multi-core CPU implementation (using the compiler directive *-O3*). The experiments consisted in running a constant number of iterations of the main loop of the algorithm. The tables show that GPGPU implementation of presented algorithm significantly outperforms its CPU counterpart. The GPGPU version is more than ten times faster then fully vectorized single core version and also significantly faster then optimized multi-core version. Table 4 describes efficiency of finding Golomb Rulers for different lengths and their percentage deviation from optimal one achieved in mentioned time. It shows that presented implementation can be used in pre-eliminary fast finding better Golomb Rulers and narrows domain for further process of optimal GR search. The simulations were run on NVIDIA Tesla m2090 and QEMU Intel 64-rhel6 (2.4 GHz). The above results were gathered when stable times of execution of algorithm were achieved in approximately five to ten executions (the observed standard deviation of the average results was negligible).

Table 3: Execution times of execution (in ms) of algorithm for Golomb ruler 13 (80 iterations of algorithm).

Population size (GR 13)	GPGPU	CPU (1 core)	CPU (-O3) (1 core)	4-cores (-O3)
64	160	3001	1315	350
128	270	4508	2604	877

6 Conclusion

The described implementation of algorithm for finding GRs shows that GPGPUs can be incorporated in process of solving some difficult black-box problems, although it is to note, that GPGPU can be efficient in implementing only some types of algorithms, namely those having

Table 4: Time of finding golomb rulers by hybrid GPGPU algorithm.

length of the ruler	time	deviation from optimal solution
4	1,5s	optimal
5	3 s	optimal
6	13min	optimal
7	40min	15%
13	1h	30-40%
14	1h	30-40%
15	1h	30-45%
16	1h	30-45%

implicitly parallel structure. On the other hand, algorithms with strong dependencies in their data flows and without data re-using or data parallelism cannot be efficiently accelerated in GPGPU (e.g. this is the case of Tabu Search algorithm).

As a main result of the presented research, efficient implementation of a hybrid system solving Golomb Ruler was presented, along with experimental results showing that our GPGPU implementation is 10 times faster than a highly optimized multicore CPU one. It must be added that proposed algorithm cannot be used for finding optimal rulers (due to local optimal solutions) but rather efficient way for narrowing solutions space. Further research will focus on incorporating simulated annealing heuristic to our GPGPU solver and building parallel hybrid GPGPU/CPU (based on OpenMP and MPI) algorithms for other difficult problems, benchmark ones and real life. We will aim also at utilizing different technologies to construct dedicated parallel frameworks, for efficient utilization of clusters of GPGPU (e.g. following the approaches presented in [23, 24]).

Acknowledgments

The research presented in the paper received partial support from AGH University of Science and Technology statutory project (no. 11.11.230.124).

References

- [1] Cuda framework. <https://developer.nvidia.com/cuda-gpus>.
- [2] Ogr project. <http://www.distributed.net/org>.
- [3] Openmp library. <http://www.openmp.org>.
- [4] G. Bloom and S. Golomb. Applications of numbered undirected graphs. In *Proceedings of the IEEE*, 65(4), pages 562–570, April 1977.
- [5] C. Cotta. Local search based hybrid algorithms for finding golomb rulers. pages 263–291, 2007.
- [6] C. Cotta, I. Dotu, A.J. Fernandez, and P. Van. A memetic approach to golomb rulers. *Parallel Problem Solving From Nature IX*, pages 255–261, 2006.
- [7] J. de Monet de Lamarck. *Zoological Philosophy: An Exposition with regard to the natural history of animals*. Macmillan, 1914.
- [8] A. Dollas, W.T. Rankin, and D. Macracken. New algorithms for golomb ruler derivation and proof of the 19 mark ruler. *IEEE Transaction on Information Theory*, 44:379–382, 1998.
- [9] I. Dotu and P. Van Hentenryck. A simple hybrid evolutionary algorithm for finding golomb rulers. *IEEE Congress on Evolutionary Computation*, 3:2018–2023, 2005.

- [10] S. Droste, T. Jansen, and I. Wegener. Upper and lower bounds for randomized search heuristics in black-box optimization. *Theory of Computing Systems*, 39:525–544, 2006.
- [11] L.J. Eshelman. *Genetic algorithms. Handbook of Evolutionary Computation*. IOP Publishing Ltd and Oxford University Press, 1997.
- [12] B. Feeney. Determining optimum and near-optimum golomb rulers using genetic algorithms. Master's thesis, University College Cork, 2003.
- [13] B. Galinier and et al. A constrained-based approach to the golomb ruler problem. In *3rd International workshop on integration of AI and OR techniques*, pages 562–570, 2001.
- [14] Vanderschel Garey. In search of optimal 20, 21 and 22 mark golomb rulers. Technical report, GVANT project, 1999. <http://members.aol.com/golomb20/index.html>.
- [15] Y. Li, K. Zhao, X. Chu, and J.Liu. Speeding up k-means algorithm by gpus. *IEEE Computer and Information Technology*, pages 115–122, jun 2010.
- [16] C.H. Lin, C.H. Liu, L.S. Chien, and S.C. Chang. Accelerating pattern matching using a novel parallel algorithm on gpus. *IEEE Transactions on Computers*, 62:1906–1916, oct 2013.
- [17] D. Merrill and A. Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21:245–272, 2011.
- [18] H. Mühlenbein. How genetic algorithms really work: I. mutation and hillclimbing. In *Proc. of 2nd Parallel Problem Solving from Nature Conference*. 1992.
- [19] M. Pietroni, P. Russek, and K. Wiatr. Accelerating select where and select join queries on gpu. *Journal of Computer Science AGH*, 14:243–252, 2013.
- [20] M. Pietroni, M. Wielgosz, D. Zurek, E. Jamro, and K. Wiatr. Comparison of gpu and fpga implementation of svm algorithm for fast image segmentation. *LNCS Springer-Verlag Heidelberg*, pages 292–302, 2013.
- [21] J.B. Shearer. Some new optimum golomb rulers. *IEEE Transaction on Information and Theory*, page 183, 1990.
- [22] S. W. Soliday, A. Homaifar, and G.L. Leebby. Genetic algorithm approach to the search for golomb rulers. *IGGA, Ed. Morgan Kaufmann*, pages 528–535, 1995.
- [23] Wojciech Turek. Erlang as a high performance software agent platform. In Dariusz Barbucha, Manh Thanh Le, Robert J. Howlett, and Lakhmi C. Jain, editors, *Advanced Methods and Technologies for Agent and Multi-Agent Systems, Proceedings of the 7th KES Conference on Agent and Multi-Agent Systems - Technologies and Applications (KES-AMSTA 2013), May 27-29, 2013, Hue City, Vietnam*, volume 252 of *Frontiers in Artificial Intelligence and Applications*, pages 21–29. IOS Press, 2013.
- [24] Wojciech Turek, Robert Marcjan, and Krzysztof Cetnarowicz. Agent-based mobile robots navigation framework. In Vassil N. Alexandrov, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part III*, volume 3993 of *Lecture Notes in Computer Science*, pages 775–782. Springer, 2006.